This paper was submitted to Transactions on Computer Systems.

Distributed System Fault Tolerance Using Sender-Based Message Logging

David B. Johnson
Willy Zwaenepoel

Department of Computer Science
Rice University
P.O. Box 1892
Houston, Texas 77251-1892

Abstract

Sender-based message logging supports transparent fault tolerance in distributed systems in which all communication is through messages and all processes execute deterministically between received messages. It uses a *pessimistic* message logging protocol that requires no specialized hardware. Sender-based message logging differs from previous message logging methods in that it logs each message in the local *volatile* memory of the machine from which it was *sent*, thus greatly reducing the overhead of message logging. Overhead is further reduced by relaxing the synchronization imposed by previous pessimistic message logging protocols. Sender-based message logging guarantees recovery from a single failure at a time in the system, and detects all cases in which multiple failures prevent recovery. Extensions are also presented to support optimistic recovery from multiple failures at once.

Sender-based message logging has been implemented under the V-System on a network of SUN-3/60 workstations. The measured overhead on V-System communication operations is about 25 percent. The overhead experienced by distributed application programs using sender-based message logging is affected most by the amount of communication performed during execution. The highest measured program overhead was under 16 percent, and for most programs, overhead ranged from about 2 percent to much less than 1 percent, depending on the problem size.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance, checkpoint/restart*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; D.4.8 [**Operating Systems**]: Performance—*measurements*

General Terms: Design, Experimentation, Measurement, Performance, Reliability

Additional Key Words and Phrases: message logging, checkpointing, recovery, volatile logging, consistent system states, V-System

---

This work was supported in part by the National Science Foundation under grants CDA-8619893 and CCR-8716914, and by the Office of Naval Research under contract ONR N00014-88-K-0140.

1 Introduction

Sender-based message logging efficiently and transparently supports fault tolerance for application programs executing in a distributed system. Processes in the system are assumed to communicate only through messages, and the execution of each process between received messages is assumed to be deterministic. Sender-based message logging guarantees recovery of a consistent system state after any failure in which only one process has failed. In all cases in which multiple processes have failed, either the system is recovered to a consistent state or the inability to recover is detected. Sender-based message logging requires no specialized hardware and adds little additional communication to the system.

With sender-based message logging, all messages received by each process are saved in a *message log*, and the state of each process is occasionally saved as a *checkpoint*. No coordination is required between the checkpointing of individual processes. When a process fails, it is recovered by restoring it from its most recent checkpoint and replaying to it from the log the sequence of messages received by it after that checkpoint. Each failed process can be recovered individually, and no surviving process is forced to roll back due to the failure. Previous fault-tolerance methods using other forms of *message logging and checkpointing* include Auros and TARGON/32 [4, 5], PUBLISHING [20], and Strom and Yemini's Optimistic Recovery [28, 27]. Sender-based message logging is unique in that it logs each message in the local *volatile* memory of the machine from which it was *sent*, as illustrated in Figure 1. Previous methods send a copy of each message to stable storage [16, 2] on disk or to a special backup process for logging. By logging messages in volatile memory, the overhead of message logging is significantly reduced.

The message logging protocol used by sender-based message logging is *pessimistic* [4, 5, 20]. A pessimistic logging protocol guarantees that after any failure, processes that have *not* failed will not be forced to roll back to complete recovery of the system. Such protocols are called pessimistic because they assume that a failure can occur at any time, and prevent processes from proceeding until this guarantee can be assured. With previous pessimistic logging protocols [4, 5, 20], message logging is synchronized with message receipt, such that a process receiving a message is not allowed to proceed until the message has been logged. Although this synchronization simplifies recovery, it can significantly increase the overhead of message logging and degrade the failure-free performance of the system. Previous pessimistic logging systems have attempted to reduce the logging overhead through the use of special-purpose hardware to assist the logging. Sender-based message logging,

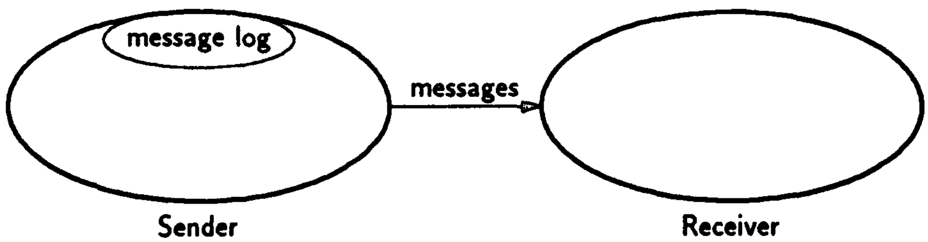

Figure 1 Sender-based message logging configuration

instead, reduces the logging overhead by logging messages in the volatile memory of the sender, and by relaxing this synchronization while still achieving the same recovery guarantee of other pessimistic logging protocols.

This paper examines the design of sender-based message logging, describes an implementation of it, and presents an analysis of its performance. More information is contained in the first author's Ph.D. dissertation [11]. Section 2 of this paper describes the model of a distributed system assumed in this work. The specification of the sender-based message logging protocol is presented in Section 3. Section 4 describes an implementation of sender-based message logging in the V-System [9, 8], and Section 5 examines its performance in this implementation. Section 6 discusses extensions to the basic sender-basea message logging protocol to guarantee recovery from multiple failures. Related work is covered in Section 7, and Section 8 presents conclusions.

2 Distributed System Model

Sender-based message logging is designed for use in existing distributed systems without the addition of specialized hardware to the system or specialized programming to applications. The following assumptions about the underlying distributed system are made:

- The system is composed of a network of fail-stop processors [24]. A fail-stop processor immediately halts whenever any failure of the processor occurs.
- Processes communicate only through messages. Messages arrive at a node asynchronously, and are queued until received by the process. Processes do not communicate via shared memory.
- The execution of each process in the system is *deterministic* [4] between received messages. That is, if two processes start in the same state and receive the same sequence of messages, they will both send the same sequence of messages and will finish in the same state. The state of a process is thus completely determined by its starting state and the sequence of messages it has received.
- The network includes a shared stable storage service [16, 2] that is always accessible to all active nodes in the system.
- The "outside world" consists of all external devices with which processes may interact, such as a time-of-day clock. Input read from the outside world is assumed to be able to be replayed durinꝗ recovery in the same order as originally received. In general, input is logged on stable storage as it enters the system, but input from read-only sources need not be logged.
- Packet delivery on the network need not be guaranteed, but reliable delivery of a packet can be achieved by retransmitting it a bounded number of times until an acknowledgement arrives from its destination.
- The network protocol supports broadcast communication. All active nodes can be reached by a broadcast through a bounded number of retransmissions of the packet.

- The underlying system is able to detect duplicate messages on arrival from the network. For simplicity of duplicate detection, we assume FIFO communication between each pair of processes. Each process tags all messages sent with a monotonically increasing ***send sequence number*** (*SSN*), and maintains a table recording the highest SSN value tagging a message received from each other process. If the SSN tagging a new message received is not greater than the current table entry for its sender, the message is considered to be a duplicate. However, FIFO communication is *not* required by the sender-based message logging protocol itself, and actual systems may use any appropriate mechanism for duplicate detection.

3 Protocol Specification

3.1 Overview

The execution of each process is divided into a sequence of ***state intervals*** by the messages that the process receives. Since process execution is deterministic between received messages, the state of a process within any state interval is a function of its state at the beginning of the interval and the contents of the message received that started the interval. Each state interval of a process is uniquely identified by a *state interval index*, which is a count of messages received by the process.

When a process sends a message, it saves a copy of the message in its local volatile memory. When the message is received, the receiver increments its own state interval index, and the *new* value becomes the index of the state interval started by the receipt of that message. This new value is also assigned as the *receive sequence number* (*RSN*) of the message. The RSN is returned to the sender to indicate the order in which this message was received relative to other messages sent to the same process, possibly by different senders. This ordering information is not otherwise available to the sender, but is required for failure recovery since these messages must be replayed to the recovering process from the log in the same order in which they were received before the failure. When the RSN arrives at the sender, it is added to the sender's volatile log with the message.

The log of messages received by a process is distributed among the processes that sent them, such that each sender has in its log only those messages that it sent. Figure 2 shows an example of such a distributed message log resulting from sender-based message logging. In this example, process Y was initially executing in state interval 6. Process Y received two messages from process X_1, followed by two messages from process X_2, and finally another message from X_1. For each message received, Y incremented its state interval index, assigned the new value as the RSN of the message received, and returned this RSN to the sender. As each RSN arrived at the sender, it was added to the sender's local volatile log with the message. After receiving these five messages, process Y is now executing in state interval 11.

During each state interval, a process may send any number of messages. Each message sent is tagged with the current state interval index of the sender. When a message is received, the receiver then ***depends on*** this state interval of the sender, since any part of the sender's state may have been included in the message. Each process records these dependencies locally in a ***dependency vector***. For each process from which this process has received messages, the dependency vector records the *maximum* state interval index tagging a message received from that process. Only the maximum

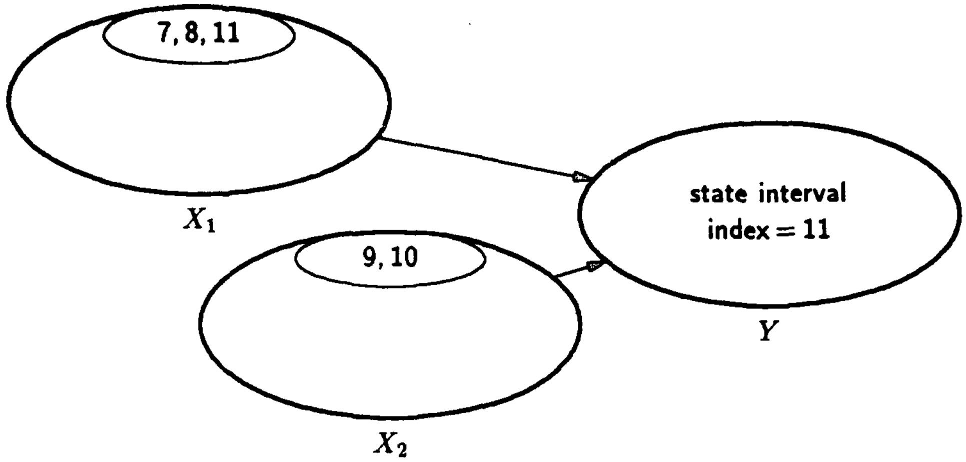

Figure 2 An example message log

index of any state interval of each other process on which this process depends is recorded, since this state interval naturally also depends on all previous state intervals of the same process.

After a failure, the system must be restored to a *consistent* system state. A system state is consistent if it *could* have occurred during the preceding execution of the system from its initial state, regardless of the relative speeds of individual processes [6]. This ensures that the total execution of the system is equivalent to *some* possible failure-free execution. During recovery, sender-based message logging uses the dependency vector maintained by each process to verify that the resulting system state that can be recovered is consistent.

3.2 Data Structures

For each process, sender-based message logging maintains a small number of data structures, as described in the list below. Except where noted, each of these data structures must be included in each checkpoint of the process, and is restored during recovery from the checkpoint. Only the most recent checkpoint of each process must be retained on stable storage. The following data structures are maintained for each participating process:

- A *state interval index*, which is incremented each time a new message is received. The *new* value becomes the index of the state interval started by the receipt of this message, and is assigned by the process as the *receive sequence number* (*RSN*) of the message. Each message sent by a process is tagged with the current state interval index of the sender.

- A *message log* of messages *sent* by the process. For each message sent, this includes the message data, the identification of the destination process, the SSN and state interval index tagging the message when sent, and the RSN returned by the receiver (which is also the index of the state interval started in the receiver by the receipt of that message). The message log is recorded in the checkpoint so that it can be restored after a failure of this process and used in any future recoveries of other processes. After a process is checkpointed, all messages

received by that process before the checkpoint may be removed from the logs in their sending processes. Only the log of messages received by each process since its most recent checkpoint must be saved in the volatile message log or in the checkpoint of each sending processes.

- A *dependency vector*, recording the maximum index of any state interval of each process on which this process currently depends. For each other process from which this process has received messages, the dependency vector stores the maximum state interval index tagging a message received from that process.

- An *RSN history list*, recording the RSN value returned for each message received by this process since its last checkpoint. For each message received, this list includes the identification of the sending process, the SSN value tagging the message, and the RSN returned by this process when the message was received. This list is used when a duplicate message is received. The RSN history list of a process is *not* included in the checkpoint. It may be purged when the process is checkpointed, since messages received by the process before this checkpoint will never be needed for recovery.

- The data structures used by the underlying system for duplicate message detection, as described in Section 2.

3.3 Message Logging

Sender-based message logging is designed to operate with any existing message transmission protocol used by the underlying non-fault-tolerant system. The following steps are required when sending a message M from process X to process Y:

1. Process X copies message M into its local volatile message log before transmitting M to process Y across the network. The message sent is tagged with the current state interval index and SSN of process X. At this point, the message is called *partially logged.*

2. Process Y receives the message, increments its own state interval index, and assigns this new value as the RSN for M. The entry for process X in Y's dependency vector is set to the maximum of its current value and the state interval index tagging message M, and an entry in Y's RSN history list is created to record this new RSN. Finally, process Y returns to process X a packet containing the RSN assigned to message M.

3. Process X adds the RSN for message M to its message log, and sends back to process Y a packet containing an acknowledgement for the RSN. Once the RSN has been added to the message log by Y, the message is called *fully logged*, or just *logged.*

After returning the RSN, process Y may continue execution without waiting for the RSN acknowledgement, but it must periodically retransmit the RSN until the acknowledgement is received or until process X is determined to have failed. Also, any new messages (including output to the outside world) sent by process Y must be delayed until the RSNs returned for all messages Y has received have been acknowledged. The operation of this protocol in the absence of retransmissions is illustrated in Figure 3.

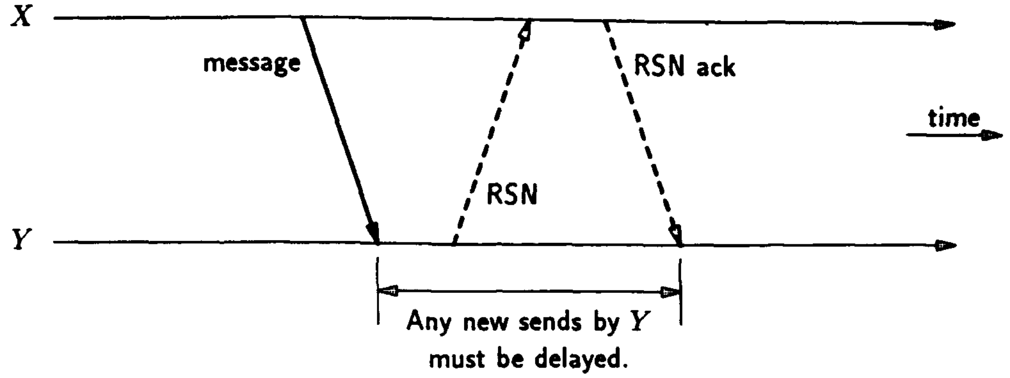

Figure 3 Operation of the message logging protocol in the absence of retransmissions

Previous pessimistic logging protocols [4, 5, 20] force each message to be logged *before it is received* by the destination process, blocking the receiver while the logging takes place. Sender-based message logging relaxes this synchronization by allowing the receiver to execute based on the message data while the logging begins asynchronously. For example, if the message requires some computation by the receiver, this computation can begin while the message is being logged. This change allows a significant decrease in the overhead of message logging, while still preserving the advantages of pessimistic message logging in terms of the simplicity of recovery. Once all returned RSNs have been acknowledged, the process knows that all messages it has received have been fully logged at their senders, and thus can be replayed during recovery in the same order in which they were originally received. By preventing the process from sending new messages before this is known, no other process can become dependent on a state of that process that may not be able to be recovered after a failure. Hence, no other process can be forced to roll back due to a failure of another process.

Processes are assumed to detect any duplicate messages on receipt, using the SSN tagging each message. When a duplicate message is received, no new RSN is assigned to the message. Instead, the receiver searches its RSN history list for an entry with the SSN tag and sending process identification of this message. If found, the RSN value there is returned to the sender. Otherwise, the receiver must have been checkpointed since originally receiving this message, and the RSN history list entry for this message has been purged. In this case, the message cannot be needed for any future recovery of this receiver, since the later checkpoint can always be used. The receiver instead returns to the sender an indication that this message need not be logged.

3.4 Failure Recovery

The following steps are used by sender-based message logging for recovery of some failed process Y:

1. The saved state of process Y is reloaded from its most recent checkpoint onto some available processor. This also restores the values of the sender-based message logging data structures described in Section 3.2.

2. All *fully logged* messages received after this checkpoint by Y are retrieved from the message logs of their sending processes. The RSN of the first message needed is one greater than the state interval index of Y recorded in the checkpoint.

3. A check is made to determine if the system state that can be recovered is consistent, using the dependency vector maintained by each process in the system. A consistent system state can be recovered if and only if no process X has an entry in its dependency vector for process Y that is greater than the RSN of the last message in the sequence of fully logged messages retrieved. This RSN gives the index of the most recent state interval of Y that can be recovered. Thus no process X depends on a state interval of Y that cannot be recreated. If a consistent system state cannot be recovered, recovery is terminated, and the system may warn the user or abort the application if desired.

4. Process Y is allowed to begin execution, but is forced to receive the retrieved sequence of fully logged messages before any other messages may be received. The fully logged messages must be received in the order of their logged RSNs. Duplicate messages sent by Y as a result of this reexecution are handled by the same method used during failure-free execution. For each duplicate message received, either the original RSN or an indication that the message need not be logged is returned to the recovering process. Sending these duplicate messages and recording the returned RSNs correctly recreates Y's volatile message log for use in any future recovery of other failed processes. Likewise, the other sender-based message logging data structures are correctly restored, since they are read from the checkpoint and are modified as a result of sending and receiving the same sequence of messages as before the failure.

5. Any *partially logged* messages destined for Y may be resent to it, along with any new messages that other processes may now need to send to Y. These messages may be sent and received in any order after the sequence of fully logged messages has been received, since any effect of their earlier receipt before the failure is not visible to any other process.

The recovery of the process is complete once the process has resent all messages it had sent before the failure and the returned RSNs have been added to the sender's message log. Any method of failure detection may be used, but failure detection must be coordinated with failure recovery. For each failed process Y, any other failure of some process Z must be detected and recovered through checking for a consistent system state (step 3 above), before Y is allowed to send any new messages after completing its own recovery. This prevents any inconsistent execution of the system if a consistent system state cannot be recovered.

If only one process has failed, a consistent system state can always be recovered, since the volatile message log at the sender survives the failure of the receiver. However, if more than one process has failed, some messages needed for recovery may not be available. For example, Figure 4 illustrates a portion of the execution of a system of three processes, in which processes 2 and 3 have failed as shown. The state recorded in the most recent checkpoint for each is indicated by a vertical bar along the execution of the process, and the index of each new state interval is indicated at the receipt of the message starting it. If message M has not been sent, a consistent system state can be recovered by sender-based message logging, since the message log of process 1 survives the failures.

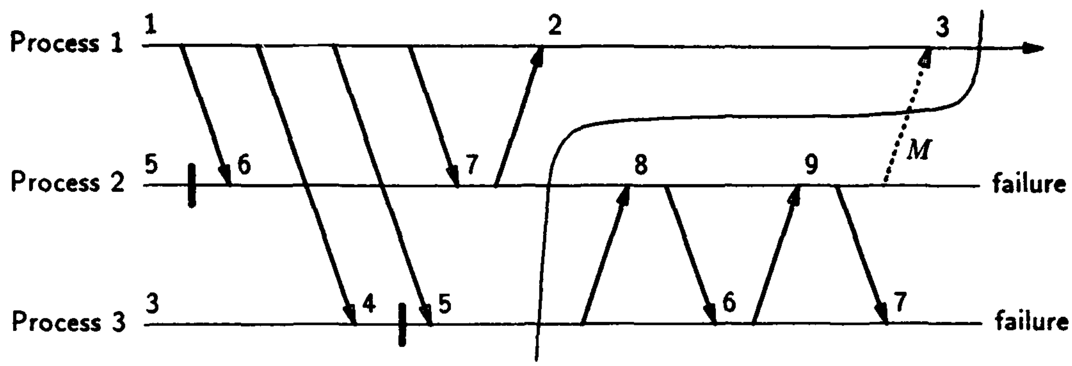

Figure 4 A multiple process failure. If message M has not been sent, sender-based message logging can recover a consistent system state.

The resulting system state is indicated by the intersection of the curve with the lines representing the execution of each process. However, if message M has been received, process 1 then depends on state interval 9 of process 2, which cannot be recreated. During the recovery of process 2, process 1 checks its dependency vector and determines that a consistent system state cannot be recovered.

To guarantee progress in the system in spite of failures, any fault-tolerance method must avoid the *domino effect* [21, 22], an uncontrolled propagation of process rollbacks necessary to restore the system to a consistent state following a failure. As with any pessimistic message logging protocol, sender-based message logging avoids the domino effect by guaranteeing that any failed process can be recovered from its most recent checkpoint, and that no other process must be rolled back during recovery.

3.5 Protocol Optimizations

The number of extra packets required for message logging can be reduced by returning more than one RSN or RSN acknowledgement in a single packet. This simple optimization is useful when an uninterrupted stream of packets is received from a single sender. The return of the RSNs is postponed until the end of the stream of packets is detected, or until a timer expires forcing their transmission, and the acknowledgements for all RSNs in a packet can be returned in a single packet. For example, when receiving a *blast* bulk data transfer [32], the RSNs for all data packets of the blast can be returned to the sender in a single packet.

Another optimization that also reduces the number of extra packets required for logging is to *piggyback* [29] RSNs and RSN acknowledgements onto existing message packets being returned, rather than transmitting them in additional special packets. The transmission of RSNs and RSN acknowledgements is postponed until a packet is returned on which to piggyback them, or until a timer expires forcing their transmission if no return packet is forthcoming. RSN acknowledgements can be piggybacked on any packet, but RSNs can be piggybacked on a packet only if *all* unacknowledged RSNs for messages received by this process are piggybacked on the same packet and are destined for the same process as the message in this packet. This preserves the correctness of

the logging protocol by ensuring that all messages received by a process will be fully logged before any new message sent by the process is seen by its destination process. When a packet is received, any RSNs and RSN acknowledgements piggybacked on it are handled before the message carried by the packet. When these RSNs are added to the message log, the messages for which they were returned become fully logged. Since this packet carries all unacknowledged RSNs from the sender, all messages received by that sender become fully logged before the new message in this packet is seen. If the RSNs are not received because the packet is lost on the network, the new message cannot be received either.

Piggybacking RSNs and RSN acknowledgements can be useful in systems in which there is frequently an existing message returned by the underlying system on which to piggyback them. For example, if the underlying message protocol uses explicit acknowledgements to ensure reliable message delivery, the RSN for the message being acknowledged can be piggybacked on the underlying acknowledgement packet. Alternatively, if a message is received that requests the application program to produce some user-level reply to the original sender, the RSN for the request message can be piggybacked on the packet carrying this reply; if the original program sends a new request to this same receiver shortly after the reply is received, the acknowledgement of this RSN, and the RSN for the reply itself, can be piggybacked on the packet carrying this new request. As long as messages are exchanged between the same two processes in this way, no new packets are necessary to return RSNs or RSN acknowledgements. When this message sequence terminates, one additional packet is needed in each direction, to return the RSN and RSN acknowledgement for the last reply message. The use of this optimization for a sequence of these request-reply exchanges is illustrated in Figure 5. This optimization is particularly useful in systems using remote procedure call [3] or other request-response protocols [8, 7], since all communication takes place as a sequence of message exchanges.

These two protocol optimizations can be combined. For example, Figure 6 illustrates the use of both optimizations with a blast bulk data transfer protocol. The RSN for every data packet of the blast can be piggybacked together on the reply packet acknowledging the receipt of the blast. If there are n packets of the blast, the unoptimized logging protocol requires an additional $2n$ packets to exchange their RSNs and RSN acknowledgements. Instead, if both protocol optimizations are combined, only one additional packet is required in each direction, to exchange the RSN and RSN acknowledgement for the packet acknowledging the blast.

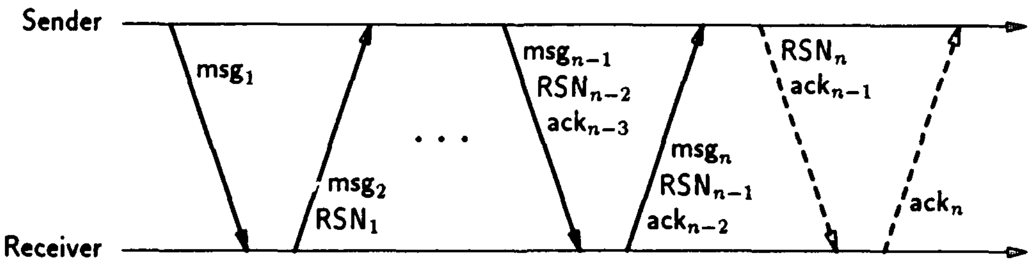

Figure 5 Piggybacking RSNs and RSN acknowledgements on existing message packets

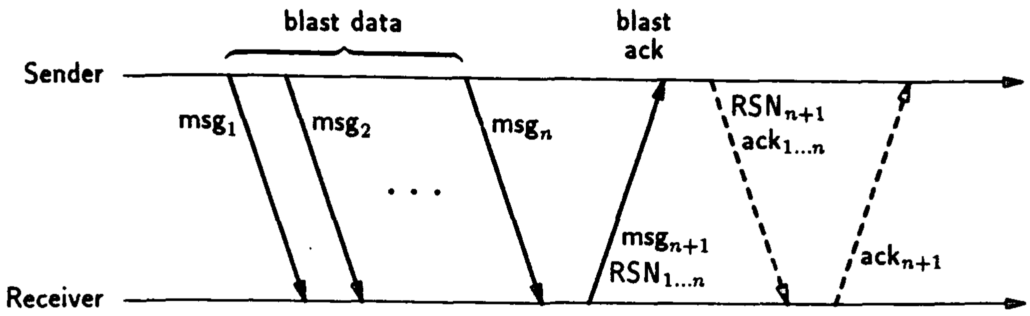

Figure 6 A blast protocol with sender-based message logging using both optimizations

Both protocol optimizations postpone the transmission of RSNs and RSN acknowledgements, which may delay the transmission of new messages that would not otherwise be delayed. If a process postpones the return of an RSN, the transmission of a new message by that process may be delayed; if the new message is not destined for the same process as the RSN, the message must be held while the RSN is sent and its acknowledgement is returned, delaying the transmission of the new message by approximately one packet round-trip time. Likewise, if a process postpones the return of an RSN acknowledgement, new messages being sent by the process expecting the acknowledgement must be delayed; in this case, the process expecting the RSN acknowledgement retransmits the RSN to force the acknowledgement to be returned, also delaying the transmission of the new message by approximately one packet round-trip time. In both cases, any possible delay is also bounded in general by the timer interval used to force the transmission of the RSN or its acknowledgement.

4 Implementation

Sender-based message logging has been implemented under the V-System [9, 8] on a collection of diskless Sun workstations connected by an Ethernet to a shared network file server. The implementation supports the full protocol specified in Section 3, including both protocol optimizations, and supports all V-System message passing operations. Although the V-System allows more than one process to share a single address space, this feature is not supported, and the implementation is limited to a single process using sender-based message logging per network node.

4.1 Division of Labor

The implementation is divided between a *logging server* process and a *checkpoint server* process running on each node in the system, and a small collection of support routines in the V-System kernel. The kernel records messages in the log in memory as they are sent, and handles the exchange of RSNs and RSN acknowledgements. This information is carried in normal V kernel packets, and is handled directly by the sending and receiving kernels. This reduces the overhead involved in these exchanges, eliminating any process scheduling delays. All other aspects of logging messages

and replaying logged messages during recovery are handled by the logging server process. The checkpoint server process manages recording checkpoints and restoring them during recovery. All logging servers in the system belong to a single V-System process group [10], and all checkpoint servers belong to a separate process group.

This use of server processes limits the increase in complexity and size of the kernel. In total, only five new primitives to support message logging and three new primitives to support checkpointing were added to the kernel. Also some changes were made to the internal operation of several existing primitives. The total size of the kernel for the SUN-3/60 configuration increased by roughly 15 kilobytes of executable instructions and 36 kilobytes of data, comprising a total increase of under 20 percent. This does not include the size of the message log in volatile memory.

4.2 Message Logging

The message log is stored in the address space of the local logging server process on each node, allowing much of the message log management to be performed by the server outside the kernel. It is organized as a list of fixed-size blocks of message logging data that are sequentially filled as needed by the kernel and are written to disk by the logging server during a checkpoint. The message log block currently being filled is always double-mapped through the hardware page tables into the kernel address space, allowing new records to be added to the log without context switching.

Each message log block is 8 kilobytes long, the same size as data blocks in the file system and hardware memory pages. Each block begins with a 20-byte header describing the extent of the space used within the block. The following two record types are used to describe the logging data in these blocks:

`LoggedMessage`: This type of record saves the data of the message sent, the SSN tagging the message, the process identifier of the receiver, and the RSN value returned by the receiver. It contains a complete copy of the packet sent, and varies in size from 92 to 1116 bytes, depending on the size of any appended data segment that is part of the message.

`AdditionalRsn`: This type of record saves an additional RSN returned for a message logged in an earlier `LoggedMessage` record. It contains the process identifier of the new receiver, the new RSN value returned, and the SSN tagging the original message. It is 12 bytes long.

Normally, only the `LoggedMessage` record type is used; the `AdditionalRsn` record type is used only for messages sent to a process group [10] or those sent as a datagram. A group message is delivered reliably to only one receiver, and unreliably to other members of the group. The first RSN returned is stored in the `LoggedMessage` record, and a new `AdditionalRsn` record is created to store the RSN returned by any other receiver of the message. Likewise, reliable delivery of a datagram message is not guaranteed by the kernel. The RSN field in the `LoggedMessage` record is not used, and an `AdditionalRsn` record is created to hold the RSN when it arrives, if the message is received.

4.3 Checkpointing

Checkpointing a process is initiated by sending a request to its local checkpoint server. This request may be sent by the kernel when the process has received a given number of messages or has consumed a given amount of processor time since its last checkpoint. Any process may also request a checkpoint at any time, but this is never necessary.

The checkpoint is written as a file on the network file server. On each checkpoint, only the pages of the user address space modified since the previous checkpoint are written to the file. The checkpoint also includes all kernel data used by the process, the state for that process in the local *team server* [8], and the state of the local logging server. This data is entirely rewritten on each checkpoint, since it is small and since modified portions of it are difficult to detect. The file server supports atomic commit of modified versions of files, and thus the most recent complete checkpoint of a process is always available, even if a failure occurs while a checkpoint is being written. To limit any interference with the execution of the process during checkpointing, most of the checkpoint data is written to the file while the process continues to execute; the process is then *frozen* while the remainder of the data is written. This is similar to the method used by Theimer for process migration in the V-System [30]. The recent checkpointing work by Li et al [17] also attempts to limit interference from checkpointing, and could be applied here as well.

Each logging server maintains a separate *message log file* on the network file server, containing a checkpoint of the message log for that node. During a checkpoint, the local logging server updates this file by writing to it all modified blocks of the message log from volatile memory. The message log file may also be updated in order to extend the amount of available space for the message log. Once a full message log block has been written to the file, it may be reused for new logging data. After the new checkpoint is complete, the group of logging servers is notified to remove all messages received by this process before the checkpoint from the log in volatile memory and from the message log file. Although reliable delivery of this notification is not ensured by the V kernel, the notification of any future checkpoint of this process will also cause their removal.

4.4 Failure Recovery

A failed process may be recovered on any available node in the network, and is restored with the same process identifier as it had before the failure. Recovery is initiated by sending a request to the checkpoint server on the node on which the process is to be recovered. Normally, the recovery request would be sent by the process that detected the failure. However, no failure detection is currently implemented, and the request instead comes from the user. Other nodes in the system determine the new network address of the recovering process through the existing V kernel network address caching mechanism.

The local logging server coordinates the replay of logged messages to the recovering process, and verifies that the resulting system state that can be recovered is consistent. The logging server sends a request to the logging server process group, giving the RSN of the first message needed for replay. The server that has this message logged returns it and all later messages that it also has logged for the recovering process; all other logging servers ignore the request. Another request is then

sent to the group, giving the RSN of the next message needed, and this procedure is repeated until all available messages have been collected. The sequence of available logged messages is complete when no reply is received from a request sent to the group after the kernel has retransmitted the request a defined number of times. The logging server then sends another message to the logging server group, giving the RSN of the last message in the sequence retrieved (or the state interval index in the checkpoint if the sequence is empty). Each logging server compares this value with the entry for the recovering process in its own dependency vector, and replies with a complaint if the dependency vector value is higher; all other logging servers do not reply. If no complaints are received after the kernel has retransmitted the request several times, the resulting system state is assumed to be consistent. The recovering process is allowed to begin execution concurrently once the first logged message has been retrieved, but until the dependency vector check has been completed, the process is not allowed to receive any messages other than those replayed. This reduces the time needed for recovery, while ensuring that no inconsistent execution occurs even if the dependency vector check fails.

This method of replaying the logged messages and checking the dependency vectors is used because no complete list of all processes in the system is maintained in the V-System, and due to the limitations of the available V-System group communication operations [10]. Retrieving the logged messages and checking the dependency vectors each terminate with a request retransmitted several times to a process group, from which no replies are received and none are expected. This avoids replies being lost on the network due to replies from other group members causing collisions on the network or buffer overflow in the receiver's Ethernet interface. Since in practice, the error rate on the Ethernet is low, these requests reach all processes and allow any replies generated to be received, with very high probability [31].

5 Performance

The performance of this implementation of sender-based message logging has been measured on a network of diskless SUN-3/60 workstations. The workstations each use a 20-megahertz Motorola MC68020 processor, and are connected by a 10 megabit per second Ethernet network to a single shared network file server. The file server runs on a SUN-3/160 using a 16-megahertz MC68020 processor, with a Fujitsu Eagle disk. This section presents an analysis of the individual costs involved with sender-based message logging in communication, checkpointing, and recovery, and an evaluation of the performance of several distributed application programs using sender-based message logging. These performance measurements were made on an otherwise idle Ethernet, and variations between individual measurements were small.

5.1 Communication Costs

Table 1 presents the time in milliseconds required for common V-System communication operations using sender-based message logging. The elapsed times required for a `Send-Receive-Reply` sequence with no appended data and with a 1-kilobyte appended data segment, for a `Send` as a datagram, and for `MoveTo` and `MoveFrom` operations of 1 and 64 kilobytes of data each were measured. These

Table 1

Performance of common V-System communication operations with sender-based message logging (milliseconds)

Operation	Message Logging		Overhead	
	With	Without	Time	Percent
`Send-Receive-Reply`	1.9	1.4	.5	36
`Send`(1K)-`Receive-Reply`	3.4	2.7	.7	26
Datagram `Send`	.5	.4	.1	25
`MoveTo`(1K)	3.5	2.8	.7	25
`MoveTo`(64K)	107.0	88.0	19.0	22
`MoveFrom`(1K)	3.4	2.7	.7	26
`MoveFrom`(64K)	106.0	87.0	19.0	22

operations were executed both with and without sender-based message logging, and the average time required for each case is shown separately. The overhead of using sender-based message logging for each operation is given as the difference between these two times, and as a percentage increase over the time without logging. These times were measured in the initiating user process, and indicate the elapsed time between invoking the operation and its completion. The overhead for most communication operations is about 25 percent.

The measured overhead reported in Table 1 is caused entirely by the time necessary to execute the instructions of the sender-based message logging protocol implementation. Because of the request-response nature of the V-System communication operations, and due to the presence of the logging protocol optimizations described in Section 3.5, no extra packets for each operation were required, and no delays in sending any message were incurred while waiting for an RSN acknowledgement to arrive. Two extra packets were required after all iterations of each test sequence to exchange the final RSN and RSN acknowledgement, but this final exchange occurred asynchronously within the kernel after the user process had completed the timing.

To better understand how this execution time is spent, the execution times for a number of components of the implementation were measured individually by executing each component in a loop a large number of times and averaging the results. The time for a single execution could not be measured directly because the hardware lacks a clock of sufficient resolution. The packet transmission overhead as a result of sender-based message logging is about 126 microseconds for messages of minimum size, including 27 microseconds to copy the message into the log. For sending a message with a 1-kilobyte appended data segment, this time increases by 151 microseconds for the additional time needed to copy the segment into the log. Of this transmission overhead, 38 microseconds occurs after the packet is transmitted on the Ethernet, and executes in parallel with reception on the remote node. The packet reception overhead is about 142 microseconds. Of

this time, 39 microseconds is spent processing any piggybacked RSNs, and 45 microseconds is spent processing any RSN acknowledgements.

These component measurements agree well with the overhead times shown in Table 1 for each operation. For example, for a `Send-Receive-Reply` with no appended data segment, one minimum-sized message is sent by each process. The sending protocol executes in parallel with the receiving protocol for each packet after its transmission on the network. The total sender-based message logging overhead for this operation is calculated as

$$2\left((126 - 38) + 142\right) = 460 \text{ microseconds.}$$

This closely matches the measured overhead value of 500 microseconds given in Table 1. The time beyond this required to execute the logging protocol for a 1-kilobyte appended segment `Send-Receive-Reply` is only the additional 151 microseconds needed to copy the segment into the message log. This closely matches the measured difference of 200 microseconds. As a final example, consider the 64-kilobyte `MoveTo` operation, in which 64 messages with 1 kilobyte of appended data each are sent, followed by a reply message of minimum size. No parallelism is possible in sending the first 63 data messages, but they are each received in parallel with the following send. After the sender transmits the last data message, and again after the receiver transmits the reply message, execution of the protocol proceeds in parallel between the sending and receiving nodes. The total calculated overhead for this operation is 18.062 milliseconds, compared with the measured overhead of 19 milliseconds.

In less controlled environments and with more than two processes communicating, communication performance may degrade because the transmission of some messages may be delayed waiting for an RSN acknowledgement to arrive. To examine the effect of this delay on the communication overhead, the average round-trip time required to send an RSN and receive its acknowledgement was measured. Without transmission errors, the communication delay should not exceed this round-trip time, but may be less if the RSN has already been sent when the new message transmission is first attempted. The RSN round-trip time required in this environment is about 550 microseconds. Although the same amount of data is transmitted across the network for a `Send-Receive-Reply` with no appended data segment, this RSN round-trip time is significantly less than the 1.4 milliseconds shown in Table 1 because the RSN exchange takes place directly between the two kernels rather than between two processes at the user level.

To examine the effect of the protocol optimizations, the performance of the same communication operations was measured again, using a sender-based message logging implementation that did not include either optimization. All RSNs and RSN acknowledgements were sent as soon as possible without piggybacking, and no packet carried more than one RSN or RSN acknowledgement. For most operations, the elapsed time increased by an average of 430 microseconds per message (packet) involved. Comparing this increase to the measured RSN round-trip time of 550 microseconds indicates that about 120 microseconds of the round-trip time occurs in parallel with other execution. This includes the time needed by the V kernel and the user process each to receive the message for which this RSN is being returned, and to form the reply message. The times for the 64-kilobyte `MoveTo` and `MoveFrom` operations and for the datagram `Send` increased by an average of only 260 microseconds per message. This increase is less, because multiple sequential messages

are sent to the same destination without intervening reply messages, and thus the sending of most messages is not forced to wait for an RSN round-trip. There is some increase, though, since each RSN and RSN acknowledgement is sent in a separate packet and must be handled separately.

5.2 Checkpointing Costs

The cost of checkpointing to the user process is small, since most data is written to the checkpoint file before freezing the process. Although the performance is highly dependent on the particular application program, the process is frozen and its execution is suspended typically for only a few tens of milliseconds.

The total elapsed time to complete the checkpoint also varies with the particular application program, and is dominated by the time required to write the modified pages of the user address space to the file. The total time is approximately 3 seconds per megabyte of modified address space, plus a small fixed cost of about 120 milliseconds. The time required to write the address space also depends on the distribution of these pages over the total address space, since only contiguous pages can be written to the checkpoint in a single operation. For each separate write operation required, the total time increases by about 3 milliseconds. A total of 17 milliseconds is required to open the checkpoint file and later close it, 0.8 milliseconds is required to checkpoint the state of the kernel, and 1.3 milliseconds is required to checkpoint the team server. The time required to checkpoint the logging server varies with the number of message log blocks to be written to the logging file, from a minimum of 18 milliseconds, and increasing by about 25 milliseconds per message log block written. For comparison, the time required to write the address space to the checkpoint is approximately 4 percent more than that required for a user process to write the same amount of data to a file on the network file server.

5.3 Recovery Costs

The time required to perform recovery is highly dependent on the particular application being recovered. This time varies most with the time required for the process to reexecute from its checkpointed state using the replayed logged messages, but this reexecution time is in general bounded by the interval at which new checkpoints are recorded. Other costs involved in recovery are similar to those involved in checkpointing.

The measured recovery time is approximately 1.5 seconds per megabyte of user address space being restored, plus a small fixed cost of about 70 milliseconds. For comparison, the time required to read the address space from the checkpoint is approximately the same as that required for a user process to read the same amount of data from a file on the network file server. This recovery time does not include the time required to retrieve the logged messages and to check the dependency vectors, since these operations occur in parallel with the reexecution of the process from its checkpointed state. The time required for the process to reexecute based on the sequence of logged messages is also not included, since this time is necessarily application-dependent.

5.4 Application Program Performance

The preceding three sections have examined the three sources of overhead caused by the operation of sender-based message logging. However, distributed application programs spend only a portion of their execution time on communication, and checkpointing and failure recovery occur only infrequently. To analyze the overhead of sender-based message logging in a more realistic environment, the performance of the following three distributed application programs, each with a different communication rate and pattern, was measured using this implementation:

nqueens: This program counts the number of solutions to the *n-queens problem* for a given number of queens n. The problem is distributed among multiple processes by assigning each a range of subproblems from an equal division of the possible placements of the first two queens. When each process finishes all allocated subproblems, it reports the number of solutions found to the main process. There is no other communication during execution. The subordinate processes do not communicate with one another, and the total amount of communication is constant for all problem sizes.

tsp: This program finds the minimum solution to the *traveling salesman problem* for a given map of n cities. The problem is distributed among multiple processes by assigning each a different initial edge from the starting city to include in all paths. A branch-and-bound algorithm is used. When each new possible solution is found by some process, it is reported to the main process, which records the minimum known solution and returns its length to this process. When a process finishes its assigned search, it requests a new edge of the graph from which to search. There is no communication between subordinate processes. Since the number of subproblems is bounded by the number of cities in the map, the total amount of communication performed is $O(n)$ for a map of n cities, but due to the branch-and-bound algorithm used, the running time is highly dependent on the map input.

gauss: This program performs *Gaussian elimination with partial pivoting* on a given $n \times n$ matrix of floating point numbers. The problem is distributed among multiple processes by giving each a subset of the matrix rows on which to operate. At each step of the reduction, the processes send their possible pivot row number and value to the main process, which determines the row to be used. The current contents of the pivot row is sent from one process to all others, and each process performs the reduction on its rows. When the last reduction step completes, each process returns its rows to the main process. All processes can communicate with all others, and the total amount of communication performed is $O(n^2)$ for an $n \times n$ matrix.

These programs were used to solve a fixed set of problems. Each problem was solved multiple times, both with and without sender-based message logging. The maps used for **tsp** and the matrices used for **gauss** were randomly generated, but were saved for use on all executions. For each program, the problem was distributed among 8 processes, each executing on a separate node of the system. When using sender-based message logging, all messages sent between application processes were logged. No checkpointing was performed during these tests, since its overhead is highly dependent on the frequency with which new checkpoints are written.

The overhead of using sender-based message logging ranged from about 2 percent to much less than 1 percent, depending on the problem size, for **nqueens** and **tsp**. The overhead for **gauss** was higher, since it performs more communication than the other programs, and ranged from about 16 percent to 3 percent. As the problem size increases for each program, the overhead decreases because the average amount of computation between messages sent increases. Table 2 summarizes the performance of these programs. The program name and problem size n are shown, together with the running time in seconds required to solve each problem, both with and without sender-based message logging. The sender-based message logging overhead for each problem is also shown in seconds and as a percentage increase over the running time without logging.

Table 3 shows the average message log sizes per node resulting from these programs. The message log sizes are also shown averaged over the elapsed execution time in seconds for each program. These message log sizes are all well within the limits of available memory on the workstations used in these tests and on other similar contemporary machines.

The effectiveness of the logging protocol optimizations was studied by examining their influence on the sending of new messages. For each message, one of three separate cases occurs. If no unacknowledged RSNs are pending (that is, all RSNs being returned have been acknowledged), the message is sent immediately with no piggybacked RSNs. If all unacknowledged RSNs can be included in the same packet, they are piggybacked on it and the message is sent immediately. Otherwise, the packet cannot be sent now and must wait for the acknowledgement of previous RSNs. The occurrences of these three cases were counted individually during the execution of each application program. Table 4 summarizes these figures as the percentage of messages sent that fall into each case, averaged over all processes. In **gauss**, piggybacking could be used less frequently than in the other two programs, since its communication pattern allowed all processes to communicate with each other during execution, reducing the probability that a message being

Table 2

Performance of the application programs using sender-based message logging (seconds)

Program	Size	Message Logging		Overhead	
		With	Without	Time	Percent
nqueens	12	5.99	5.98	.01	.17
	13	34.61	34.60	.01	.03
	14	208.99	208.98	.01	.01
tsp	12	5.30	5.19	.11	2.12
	14	16.40	16.13	.27	1.67
	16	844.10	841.57	2.53	.30
gauss	100	12.41	10.74	1.67	15.55
	200	71.10	66.40	4.70	7.08
	300	224.06	217.01	7.05	3.25

Table 3

Message log sizes for the application programs using sender-based message logging (average per node)

Program	Size	Total		Per Second	
		Messages	Kilobytes	Messages	Kilobytes
nqueens	12	8	1.9	1.30	.32
	13	8	1.9	.23	.06
	14	8	1.9	.04	.01
tsp	12	43	5.5	8.09	1.04
	14	48	6.1	2.91	.37
	16	59	7.3	.07	.01
gauss	100	514	95.4	41.44	7.69
	200	1113	292.8	15.66	4.12
	300	1802	593.7	8.04	2.65

sent is destined for the same process as the pending unacknowledged RSNs. For the **nqueens** and **tsp** programs, piggybacking utilization was lower in the main process than in the others, due to the differences in their communication patterns. For all programs, though, more than half the messages could be sent without waiting.

Because the logging protocol optimizations may postpone the return of an RSN acknowledgement, some messages that could be sent immediately without these optimizations may instead be delayed before sending. To evaluate this effect, the application programs were reexecuted using a sender-based message logging implementation that did not include either optimization; all RSNs and acknowledgements were returned immediately and no piggybacking was performed. The statistics reported in Table 4 were again measured, except that only two cases were now possible: messages sent while no unacknowledged RSNs were pending, and messages forced to wait for RSN acknowledgements. In these measurements, the percentage of messages sent with piggybacked RSNs shown in Table 4 were instead divided approximately equally between the two other cases. Although any actual delays caused by the protocol optimizations could not be measured directly, these measurements indicate that such extra delays do not commonly occur. The effectiveness of piggybacking as an optimization has also been demonstrated recently in another context by Joseph and Birman [14].

The additional overhead caused by checkpointing depends on the frequency with which new checkpoints are created. To evaluate this overhead, each application program was reexecuted to solve its largest problem, with new checkpoints written by each process after each 15 seconds of processor time. A high checkpointing frequency was used in order to generate a significant amount of checkpointing activity to be measured. For **nqueens** and **tsp**, the additional overhead from this level of checkpointing was less than 0.5 percent of the required running time for that application with sender-based message logging. For **gauss**, checkpointing overhead was about 2 percent. This

Table 4

Statistics on message sending by the application programs (percentage of messages sent)

Program	Size	No RSNs Pending	RSNs Piggybacked	Wait For RSN Ack
nqueens	12	42.9	44.4	12.7
	13	41.3	46.0	12.7
	14	41.3	46.0	12.7
tsp	12	19.9	62.4	17.6
	14	22.3	63.5	14.2
	16	33.0	58.3	8.7
gauss	100	20.7	30.0	49.2
	200	29.4	28.6	42.0
	300	29.0	31.0	40.0

is higher than for the other two programs because **gauss** modifies more data during execution, which must be written to the checkpoint.

6 Multiple Failure Recovery

As described, sender-based message logging cannot recover a consistent system state in some cases in which more than one process has failed at a time. However, sufficient information is present in the existing process checkpoints to allow recovery in many cases of multiple failures not supported by the basic recovery procedure. For example, Figure 4 of Section 3.4 showed an example system state (with message M having been received) that cannot be recovered by the basic sender-based message logging mechanism. Here, process 1 depends on an unrecoverable state interval of process 2, because of its receipt of message M. The recovery procedure can be extended to recover a consistent system state in this case if the existing checkpoint of process 1 records the state of the process before it received M. Process 1 is simply rolled back by forcing it to fail and recovering it using this checkpoint. Message M is not replayed to process 1 during this recovery, making the recovered state of process 1 consistent with the recovered states of processes 2 and 3.

With this extension, each such surviving process X that depends on some unrecoverable state interval of some failed process Y must be rolled back. This dependency is detected during the recovery of Y as described in Section 3.4, using the dependency vectors maintained by each process. If the current checkpoint for process X was written before the message from Y was received that caused this dependency, then process X is rolled back. To preserve as much of the existing volatile message log as possible, each such process X is rolled back one at a time after the reexecution of the original failed processes is completed. As the original failed processes reexecute using the sequences of messages that can be replayed, they resend any messages they sent before the failure, and thus recreate much of their original volatile message log that was lost from the failure. Then, as each of

these additional processes is forced to fail and is recovered, it will recreate its volatile message log during its reexecution as well. By rolling these processes back one at a time, no additional logged messages needed for their reexecution from their checkpointed states will be lost.

If the checkpoint for some process X that must be rolled back was not written early enough to allow the process to roll back to before the dependency on process Y was created, recovery of a consistent system state using its existing checkpoint is not possible. To guarantee recovery of a consistent system state in this case, sender-based message logging can be extended further to retain on stable storage all checkpoints for all processes, rather than saving only the most recent one for each process. Then, the existence of a checkpoint for each such process X that can be used to roll X back far enough is ensured.

These extensions to the recovery procedure are *optimistic* [28, 13, 25], since they may force surviving processes to also roll back in order to recover after a failure. The volatile message log is used for recovery when possible, and the saved copy of the message log in each process checkpoint is used as an optimistic message log otherwise. Although not all checkpoints must be retained on stable storage to guarantee recovery with these extensions, determining which checkpoints can safely be removed is a separate problem, requiring an additional protocol or algorithm as with existing optimistic message logging methods [28, 13, 25]. The domino effect is still avoided by these extensions, since the data in the checkpoints is not volatile. There is always a unique maximum recoverable system state using the checkpointed message logs, which never decreases [13]. No process may be forced to roll back beyond this state. If each process eventually records new checkpoints, this maximum recoverable system state must eventually increase. By using the surviving volatile message logs as well, process roll back is further reduced.

7 Related Work

Many fault-tolerance systems require application programs to be written according to specific computational models to simplify the provision of fault tolerance. For example, the Argus [18, 19] and Camelot [26] systems require applications to be structured as a set of atomic actions on abstract data types. Likewise, some systems, such as the Tandem NonStop system [1], require the programmer to embed fault-tolerance support into each application. Since sender-based message logging is transparent, it does not impose such restrictions on application programs.

Sender-based message logging differs from other message logging protocols primarily in that messages are logged in the local *volatile* memory of the *sender*. Sender-based message logging is also unique among existing *pessimistic* message logging protocols [4, 5, 20] in that it requires no specialized hardware to assist with logging. The TARGON/32 system [5], and its predecessor Auros [4], log messages at a backup node for the receiver, using specialized networking hardware that provides three-way atomic broadcast of each message. With this networking hardware assistance and using available idle time on a dedicated processor of each multiprocessor node, the overhead of providing fault tolerance in TARGON/32 has been reported to be about 10 percent [5]. Sender-based message logging causes less overhead for all but the most communication-intensive programs, without the use of specialized hardware. The PUBLISHING mechanism [20] proposes

the use of a centralized logging node for all messages, which must reliably receive every network packet. Although this logging node avoids the need to send an additional copy of each message over the network, providing this reliability guarantee seems to be impractical without additional protocol complexity [23]. Strom and Yemini's Optimistic Recovery mechanism [28] logs all messages on stable storage on disk, but Strom, Bacon, and Yemini have proposed enhancements to Optimistic Recovery, using ideas from sender-based message logging, to avoid logging some messages on stable storage [27].

Some of the simplicity of the sender-based message logging protocol results from the limitation of guaranteeing recovery from only a single failure at a time. This allows the messages to be logged in volatile memory, significantly reducing the overhead of logging. Similar single-failure assumptions were also made by Tandem NonStop, Auros, and TARGON/32, but without achieving such a reduction in fault-tolerance overhead. The addition of the extensions of Section 6 to handle multiple failures causes no additional overhead during failure-free operation, although to guarantee recovery requires that all checkpoints be retained on stable storage. Also, the recovery from multiple failures at once using these extensions may require longer to complete than with other methods, since any processes other than those that failed that must be rolled back, must do so one at a time.

Optimistic message logging methods [28, 13, 25] have the potential to outperform pessimistic methods, since message logging proceeds asynchronously without delaying either the sender or the receiver for message logging to complete. However, these methods require significantly more complex protocols for logging, since each process must essentially be notified of the progress of the logging of messages received by each other process. Also, failure recovery in these systems is more complex and may take longer to complete, since processes other than those that failed may need to be rolled back to recover a consistent system state. Finally, optimistic message logging systems may require substantially more storage during failure-free operation, since logged messages may need to be retained longer, and processes may be required to save more than just their most recent checkpoint. Sender-based message logging achieves some of the advantages of asynchronous logging more simply by allowing messages to be received before they are fully logged.

Message logging and checkpointing methods differ from those using *global checkpointing* [6, 15] in that separate processes can be checkpointed individually, evening the load on the network and file server on which checkpoints are recorded. With global checkpointing, the network or file server may become a performance bottleneck, and the coordination required between processes during checkpointing may significantly add to the overhead of the system. Global checkpointing has the advantage of not requiring process execution to be deterministic, and can support recovery from any number of concurrent failures. However, a global checkpoint must be created each time before output from the system can be released to the outside world. Message logging and checkpointing avoids this expense by using the logged messages to allow states of a process between its checkpointed states to be restored. Global checkpointing methods could be used to limit the number of checkpoints that must be retained for each process with the multiple failure recovery extensions of Section 6, but this would increase checkpointing overhead during failure-free execution.

This work improves on our earlier work with sender-based message logging, which was reported before the system had been implemented [12]. The protocol optimizations of Section 3.5 result in a

significant reduction in the number of extra network packets required for message logging. Sender-based message logging now also detects all cases in which the system cannot be recovered to a consistent state following a failure. Furthermore, the extensions of Section 6 allow the system to be recovered in these cases of multiple failures, although they may also require surviving processes to be rolled back during recovery.

8 Conclusion

Sender-based message logging is a transparent method of providing fault tolerance in distributed systems in which all process execution is deterministic and all process communication is through messages. It uses *pessimistic* message logging and checkpointing to record information for recovering a consistent system state following a failure. It differs from previous message logging protocols in that each message is logged in the local *volatile* memory of the node from which it was *sent*. The order in which the message was *received* relative to other messages sent to the same receiver is required for recovery, but this information is not usually available to the message sender. With sender-based message logging, when a process receives a message, it returns to the sender a *receive sequence number* (*RSN*) to indicate this ordering information. When the RSN arrives at the sender, it is added to the local volatile log with the message. To recover a failed process, it is restarted from its most recent checkpoint, and the sequence of messages received by it after this checkpoint are replayed to it in ascending order of their logged RSNs.

Sender-based message logging concentrates on reducing the overhead placed on the system from the provision of fault tolerance by a pessimistic logging protocol. The cost of message logging is the most important factor in this system overhead. Keeping the message log in the sender's local volatile memory avoids the expense of synchronously writing each message to disk or sending an extra copy over the network to some special logging process. Overhead is further reduced by relaxing the synchronization imposed by previous pessimistic message logging protocols. Unlike previous pessimistic logging protocols, sender based message logging requires no specialized hardware to assist with logging. Since the message log is volatile, sender-based message logging can guarantee recovery from only a single failure at a time within the system. In all cases in which multiple processes have failed, either the system is recovered to a consistent state or the inability to recover is detected. Extensions to the basic sender-based message logging protocol also guarantee recovery in all cases of multiple failures.

Performance measurements from a full implementation of sender-based message logging under the V-System verify the efficient nature of this protocol. Measured on a network of SUN-3/60 workstations, the overhead on V-System communication operations is approximately 25 percent. The overhead experienced by distributed application programs using sender-based message logging is affected most by the amount of communication performed during execution. For Gaussian elimination, the most communication-intensive program measured, this overhead ranged from about 16 percent to 3 percent, for different problem sizes. For the other programs measured, overhead ranged from about 2 percent to much less than 1 percent.

Acknowledgements

We would like to thank Rick Bubenik, John Carter, Elmootazbellah Nabil Elnozahy, Jerry Fowler, and Mark Mazina for their comments on earlier drafts of this paper. We are also grateful to Ken Birman, David Cheriton, Ed Lazowska, and Rick Schlichting, who provided many useful comments on this work, and to the referees, whose suggestions helped to improve its presentation.

References

[1] Joel F. Bartlett. A NonStop kernel. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 22–29. ACM, December 1981.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[3] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 90–99. ACM, October 1983.

[5] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[6] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[7] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of the SIGCOMM '86 Symposium: Communications Architectures & Protocols*, pages 406–415. ACM, August 1986.

[8] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[10] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[11] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. Ph.D. thesis, Rice University, Houston, Texas, December 1989.

[12] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The Seventeenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 14–19. IEEE Computer Society, June 1987.

[13] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[14] Thomas A. Joseph and Kenneth P. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1):54–70, February 1986.

[15] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[16] Butler W. Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, Palo Alto, California, April 1979.

[17] Kai Li, Jeffrey F. Naughton, and James S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, pages 79–88. ACM, March 1990.

[18] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[19] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122. ACM, November 1987.

[20] Michael L. Powell and David L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109. ACM, October 1983.

[21] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[22] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[23] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[24] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[25] A. Prasad Sistla and Jennifer L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing.* ACM, August 1989.

[26] Alfred Z. Spector. Distributed transaction processing and the Camelot system. In *Distributed Operating Systems: Theory and Practice*, edited by Yakup Paker, Jean-Pierre Banatre, and Müslim Bozyiğit, volume 28 of *NATO Advanced Science Institute Series F: Computer and Systems Sciences*, pages 331–353. Springer-Verlag, Berlin, 1987.

[27] Robert E. Strom, David F. Bacon, and Shaula A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49. IEEE Computer Society, June 1988.

[28] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[29] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[30] Marvin N. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-System. In *Proceedings of the Tenth ... Symposium on Operating Systems Principles*, pages 2–12. ACM, December 1985.

[31] Willy Zwaenepoel. *Message Passing on a Local Network.* Ph.D. thesis, Stanford University, Stanford, California, October 1984.

[32] Willy Zwaenepoel. Protocols for large data transfers over local area networks. In *Proceedings of the 9th Data Communications Symposium*, pages 22–32. IEEE Computer Society, September 1985. Also reprinted in *Advances in Local Area Networks*, edited by Karl Kümmerle, John O. Limb, and Fouad A. Tobagi, Frontiers in Communications series, chapter 33, pages 560–573, IEEE Press, New York, 1987.

INDEX TO TECHNICAL REPORTS:

1. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing", by D.B. Johnson and W. Zwaenepoel, Rice COMP TR88-68.

2. "Distributed System Fault Tolerance Using Message Logging and Checkpointing", by D.B. Johnson, Ph.D. Thesis, Rice COMP TR89-101.

3. "Causal Distributed Breakpoints", by J. Folwer and W. Zwaenepoel, RICE COMP TR90-107.

4. "Computer Systems Research at Rice University", Annual Report 1989-90.